RMIT UNIVERSITY Computer Science and Information Technology

RMIT School of Computer Science and Information Technology

Programming 2

Topic 5: Graphical User Interfaces I
Introduction, Event Handling

Lecture Slides

Graphical User Interfaces - Introduction

Most (modern) application programs (applications or apps) do not communicate with their users solely via text input and output. They use a **Graphical User Interface (GUI)**.

Most users find a **graphical user interface more convenient and easier to use** than a **text-mode console** system (recognition versus recall).

A **Graphic User Interface** (GUI, pronounced 'gooey') allows interaction via windows containing buttons, menus, control panels, dialog boxes, etc.

However, it is important to note that a **good user interface cannot compensate for a poor program.**

Likewise, an otherwise excellent program **can be rendered unusable or useless** by a poor user interface (see following slide, usability).

Therefore, it is important to design an **effective means** for the **user to interact** with the program.

Designing for Usability

Usability is a combination of the following user-oriented characteristics:

(Designing the User Interface 3rd Ed., Shneiderman, B., 1998)

- High speed of user task performance.
- Low user error rate.
- Ease of learning.
- User retention over time.
- Subjective user satisfaction.

The user is the final judge of the usability and appropriateness of the interface.

Achieving usability requires attention to two main components:

- The product (interactive software system)
- The process by which the product is developed.

Is primarily the domain of usability specialists and graphic designers. This course emphasis quality of the user interface ***code***

Java's GUI

Java's Abstract Window Toolkit (AWT) and Swing are standard Java libraries of packages and classes designed to provide a framework for programmers to create GUIs. AWT provides the basic GUI classes. Swing extends AWT and provides more advanced GUI components. They share the same concepts and event model. More on Swing in the next slide.

Where available Swing is preferred over AWT however AWT is still used for 2D graphics, fonts, color, layout management and event handling (since Swing does not replace or supersede these AWT features).

This course:

- Covers general software design principles
- Introduces AWT (especially the core event handling and layout management model)
- Introduces basic Swing components for designing the graphical user interface

NOTE: IBM provides an alternative GUI framework called SWT (Standard Widget Toolkit) which is not covered in this course (Eclipse is a good example of an SWT project).

Swing Basics

Swing is an additional graphical interface toolkit (distributed as part of Java SE since Java 1.2) that provides many improvements over AWT that was introduced with Java 1.0 and refined with Java 1.1.

Swing is built on top of the AWT component set, and relies on it for its basic functionality.

Swing and AWT components, however, should not be used together in the same interface (due to z-order incompatibility).

The components and other classes related to AWT are in the series of packages named *java.awt.**

The components and other classes related to Swing are in the series of packages named *javax.swing.**

Swing UI components begin with 'J' to differentiate them from their AWT counterparts; e.g. the Swing button object is JButton, the AWT equivalent is Button.

Lightweight and Heavyweight Components

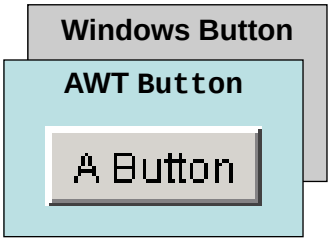
Most Swing components are "lightweight" components, compared to the AWT (where they are "heavyweight".)

Lightweight components are those that are written completely in Java, and do not rely at all on the host operating system.

Heavyweight components depend on the related 'peer' component of the native operating system.

This is why AWT-based programs look slightly different on each platform, but Swing programs maintain the same look (and also why these component sets should not be mixed together.)

The GUIs we develop in this course will be Swing based.



Heavyweights rely on their native peers.

Applets and Applications

There are **two kinds** of program in Java SE: **applets** and **applications.**

Applets are **container-based** applications which usually **run in a web browser.** An applet is typically downloaded from the network (or internet) by a web browser to execute locally on your computer. There are a number of (mainly security related) **restrictions that are placed on applets** (discussed later). They are always GUI based.

Java applications can be console (text) or GUI based**. GUI applications** are **standard Java programs** which create a graphical user interface. These can be run from the command prompt and are sometimes called **stand-alone** applications.

The process of creating a graphical user interface for applets and applications is very similar and both use the same AWT and Swing components.

JApplets and Basic Graphics

JApplets (Swing applets) are created by extending the JApplet class:

```
public class MyApplet extends JApplet
```

There are a number of methods (see slide 12 for details) inherited from JApplet which can be overridden to provide custom behaviour e.g.:

init() – this is the place for initialising the applet.
paint() – this controls the (re-)drawing of the applet.

The paint() or paintComponent() method can be used simply to make the applet a drawing canvas, using the predefined Java drawing methods of the Graphics class. However more often AWT or Swing components are used to create the user interface (especially for business type applications cf. games).

Note that JApplet extends java.applet.Applet (AWT Applet) and thus much of its functionality comes from AWT.

Swing Applets (1)

Let's compare a simple text I/O application with a GUI applet. Recall the famous HelloWorld program below.

```
public class HelloWorld {

  public static void main(String[] args) {

    System.out.println("Hello world");
  }
}
```

Calling method (function) ***println*** on the attribute ***out*** of the class ***System***.

NB. ***out*** is an ***OutputStream*** object. ***System.out*** represents the standard output (console).

Let's then do the same as a GUI …

Swing Applets (2)

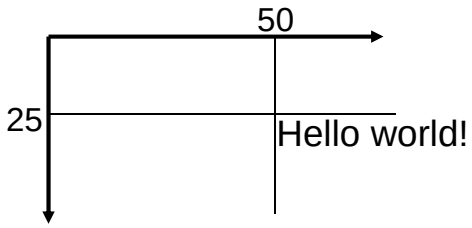
```
import javax.swing.JApplet;
import java.awt.Graphics;

public class HelloWorld extends JApplet {

  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

Import from swing and awt packages

Extend JApplet class

"Automatic" container based method; no main() method in an applet.

Window coordinates in pixel units – (0,0)



Running Applets

Save applet code in a file, eg. HelloWorld.java

Compile to produce HelloWorld.class

Save below in a file: HelloWorld.html

```
<HTML>
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET></HTML>
```

Open a browser and load HelloWorld.html.

This will run the applet. (Browser must be enabled to run Java)

Alternatively, open ***appletviewer*** that comes with Sun's free JDK.

The compiled .class file is downloaded (from across the Internet if necessary, depending on the path given in Applet Code tag) and interpreted in the local computer by the browser.

Applets – Swing and AWT

Older browsers may not support Swing components. In this case you need to install the Java plug-in (see http://java.sun.com/products/plugin/)

As far as this course is concerned, it is sufficient to test your applets entirely in *appletviewer*

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorld extends Applet {

  public void paint(Graphics g) {
    g.drawString("Hello world!", 50, 25);
  }
}
```

In order to write Hello World as an AWT applet, some minor changes are necessary.

The Applet Life Cycle

Method Name	Executed by browser when an HTML document that contains the applet...	Is often used to...
init	Is loaded by the browser	Set up user-interface, start conditions
start	Is displayed by the browser or revisited and displayed	Re-initialise suspended functionalities
stop	Is closed by the browser or the browser is shut down	Suspend (resource-consuming) functionalities
destroy	After stop	Free up resources

Method Name	Executed by browser when the applet need redrawn, the named method...	Is often used for...
repaint	Redraws the applet. Calls method *update*	Advanced drawing and animation techniques
update	Clears the applet's graphics area. Calls method *paint*	Advanced drawing and animation techniques
paint	Redraws the applet graphics area	Simple drawings

Another Example: Bouncing Ball Applet (1)

```
import javax.swing.JApplet;
import java.awt.*;

/* <APPLET CODE="Ball.class" HEIGHT=300 WIDTH=300>
   </APPLET> */

public class Ball extends JApplet {
  private int x = 0, y = 0, dx = 1, dy = 2;
  private int diameter = 50, width = 0, height = 0;

  public void init() {
    width = getSize().width;
    height = getSize().height;
  }
```

The commented-out HTML code is a short cut to run an applet through appletviewer. No need for an html file. Simply type:

% appletviewer Ball.java

This code gets the size of the applet surface.

Another Example: Bouncing Ball Applet (2)

```
  public void paint(Graphics g){
    while (true) {
      for (int i=0; i<10000; i++);
      x += dx;
      if (x < 0) dx = 1;
      if (x > width-diameter) dx = -1;
      y += dy;
      if (y < 0) dy = 2;
      if (y > height-diameter) dy = -2;
      g.setColor(this.getBackground());
      g.fillRect(0,0,width,height);
      g.setColor(Color.red);
      g.fillOval(x,y,diameter,diameter);
    }
  }
}
```

this loop will provide a short delay.

this code will clear the graphics area.

Applet Security

For security purposes, under normal circumstances, applets run in a sandbox which provides them limited access to the machine;

for example, an applet cannot read or write the local file system or communicate with hosts other than that from which it was downloaded.

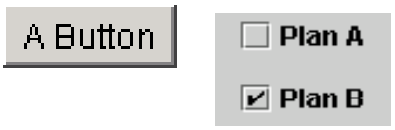
In special cases where these security provisions need to be overruled, the applet can be run with options which specify the actions to be allowed.

Often this is accompanied by a signed certificate to verify the applet provider (e.g. VeriSign certificates) however this is beyond the scope of this course.

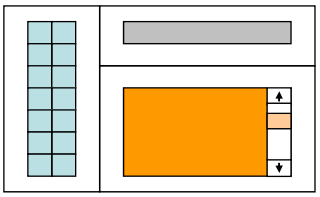
Developing a GUI

There are **three basic things** that are necessary to develop a graphical user interface:

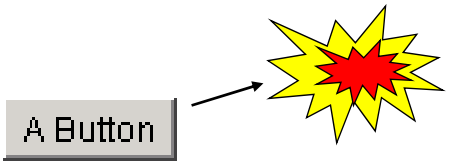
The GUI components: these are the buttons, labels, check boxes, etc. which form the interface.



Component arrangement: this is the scheme whereby the UI components are arranged to create the interface.



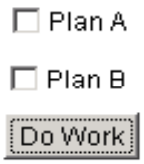
Response to user requests: this is the act of associating actions to user requests (known as 'events'.)



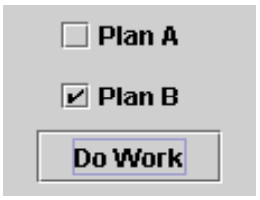
AWT and Swing Components

There are two sets of components in Java: the **AWT** (Abstract Windowing Toolkit), and the **Swing component set**.

The **AWT component set** is a very **basic set** of user interface items.



The **Swing component set** (which is an extension of the AWT set) are **far richer** in presentation and functionality, however they are **only available in later Java versions.** Most web browsers **cannot use Swing** without plug-ins; many applets use AWT.



Mixing AWT and Swing components **is not recommended!**

Components and Containers

The AWT set is structured in a class hierarchy; **every UI component** is a **descendant of the `Component` class**. This contains the **very basic functionality** common to all components, such as the **ability to be laid out** on an interface, **foreground and background colours**, etc.

All Swing components (both atomic and non-atomic) are derived from the class **java.awt.Container**.

All Swing components whose names begin with "**J**", except for the top-level containers, are derived from **javax.swing.JComponent**

A **Container** is a special type of component to which you can add other components. Example, JFrame, JApplet, JPanel etc. All containers have methods to add and remove components

```
public Component add (Component comp);
public void remove (Component comp);
```

as well methods to arrange these components (more on this next lecture)

The Swing Containment Hierarchy

A Swing containment hierarchy typically has at least:

a top-level container: the root container that holds everything together. e.g. JApplet or JFrame

one or more intermediate containers to simplify the positioning of atomic components. e.g. JPanel, JScrollPane

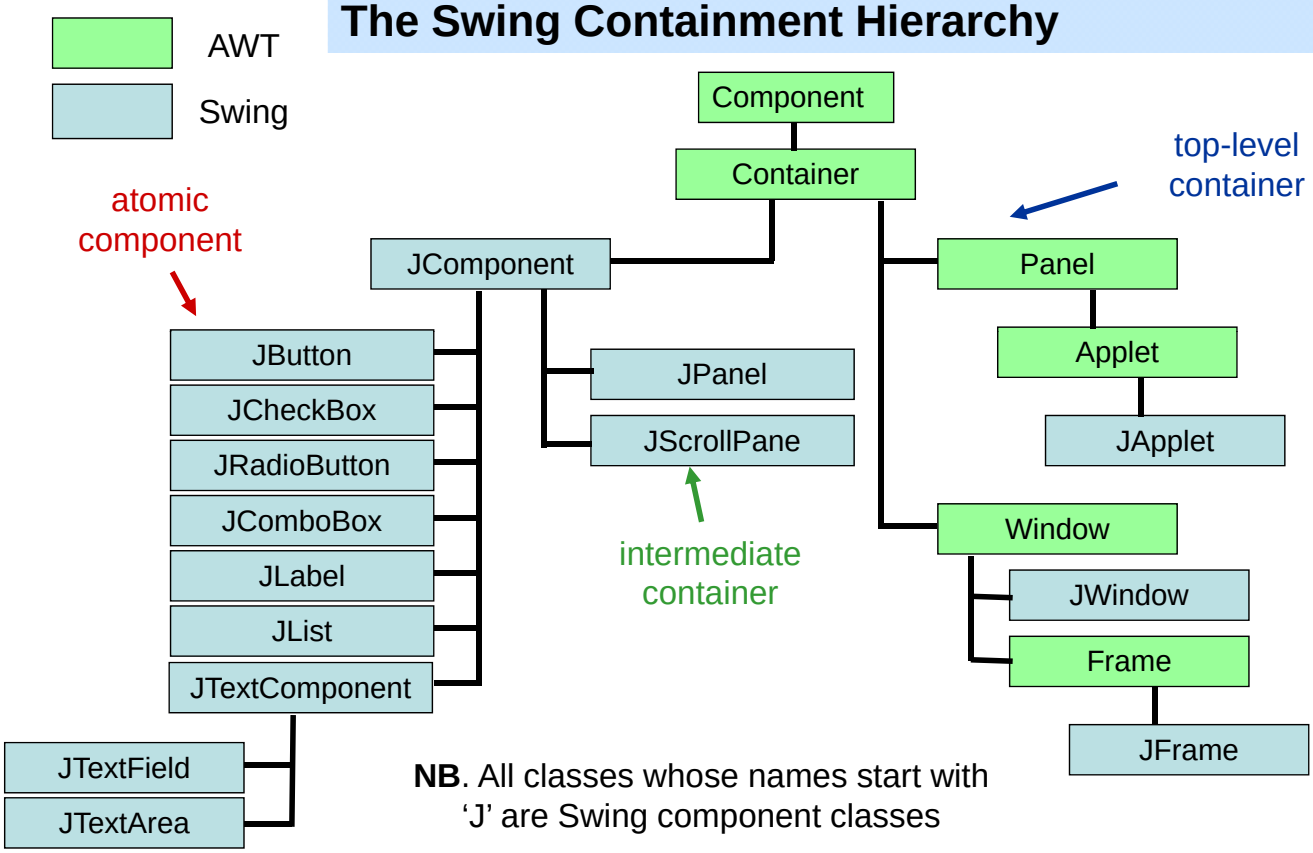
an atomic component: a self-sufficient component not holding other components. eg. JButton, JLabel

In a Swing application:

The top-level container must be Swing. e.g, JApplet (not Applet)

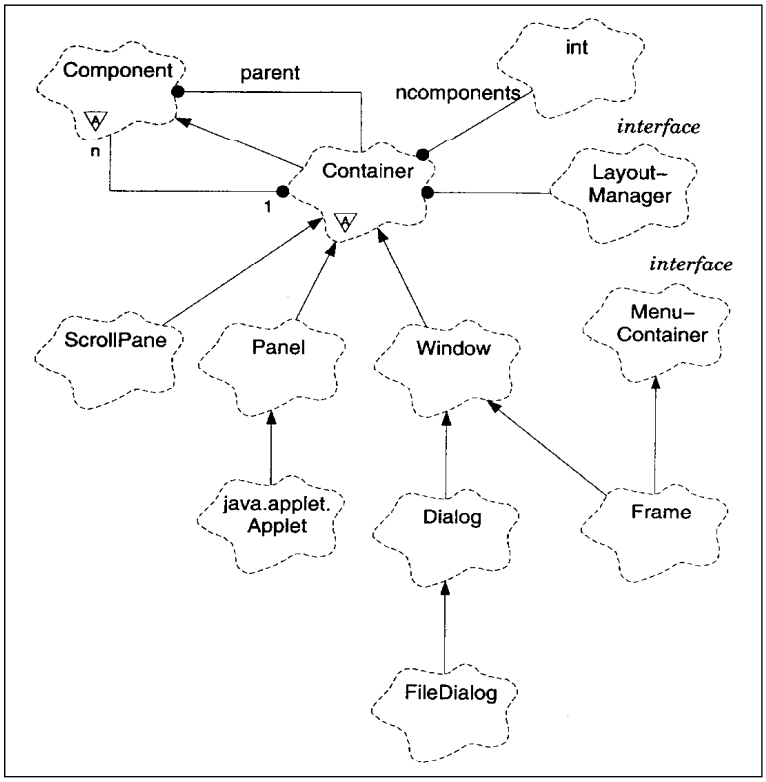
A component cannot be added directly to the top-level container, but must be added to an intermediate container contained in the top-level container, called the content pane. This is done by calling the method public Container. getContentPane() that returns a reference to its content pane (which is actually a JPanel).

The Swing Containment Hierarchy



NB. All classes whose names start with 'J' are Swing component classes

Class Diagram – Component / Container Relationships



Hello World, Revisited

The simple “Hello world!” program, implemented using a **`JLabel`** GUI component for the text:

```
import javax.swing.*;

public class HelloWorld extends JFrame {

  JLabel greeting = new JLabel("Hello world!");

  public HelloWorld() {
    Container c = getContentPane();
    c.setLayout(new FlowLayout());
    c.add(greeting);
    setVisible(true);
  }
  public static void main(String args[])
  {
    new HelloWorld();
  }
}
```

Top-level Container

Atomic Component

Intermediate Container

GUI Component Example (1)

```
import java.awt.*;
import javax.swing.*;
public class GUIComponents extends JFrame {

  JPanel pn;
  JButton button = new JButton("Button");
  JCheckBox checkBox = new JCheckBox();
  JLabel label = new JLabel("Label");
  JTextField textField = new JTextField("Text Field");
  JTextArea textArea = new JTextArea("Text Area", 5, 20);

  public void GUIComponents() {
    super();
    Container container = getContentPane();
    container.setLayout(new FlowLayout());
    container.add(button);
    container.add(checkBox));
    container.add(label);
```

GUI Component Example (2)

```
pn = new JPanel();
pn.setBackground(Color.red);
container.add(pn);
container.add(textField);
container.add(textArea);
DefaultListModel data = new DefaultListModel();
data.addElement("item 1"); data.addElement("item 2");
data.addElement("item 3"); data.addElement("item 4");
data.addElement("item 5"); data.addElement("item 6");
data.addElement("item 7"); data.addElement("item 8");
data.addElement("item 9"); data.addElement("item 10");
JList list = new JList(data);
JScrollPane scroll = new JScrollPane(list);
container.add(scroll);
JComboBox selector = new JComboBox();
selector.addItem("Yes");
selector.addItem("No");
selector.addItem("Maybe");
container.add(selector);
```

This code sets up the list.

This code sets up a drop-down box.

GUI Component Example (3)

```
JRadioButton oft = new JRadioButton("often", true);
JRadioButton st = new JRadioButton("sometimes");
JRadioButton never = new JRadioButton("never");
ButtonGroup group = new ButtonGroup();
group.add(oft);
group.add(st);
group.add(never);

container.add(oft);
container.add(st);
container.add(never);
  }
}
```

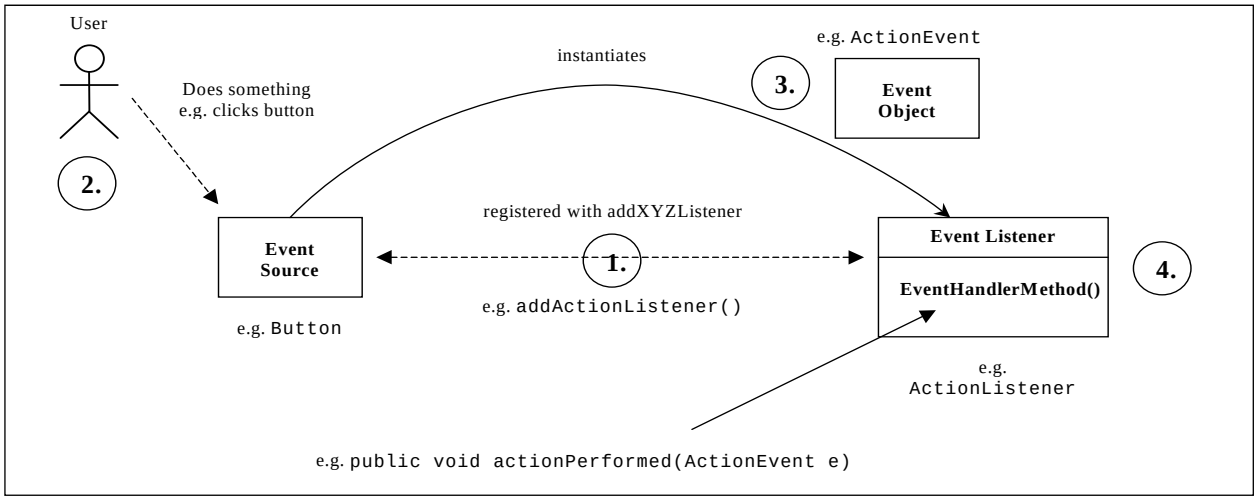
A ButtonGroup ensures only one item in the group can be selected.

AWT - Sources, Listeners and Events

- Delegation event model in AWT 1.1 (and later)
  - Event sources (usually `Components`) fire *events*
  - Events are listened for and acted upon by *event listeners*
  - Listeners are registered with a source by invoking the component's `addXYZlistener(XYZlistener)` method
  - Can add multiple listeners to an event source
    - Notification order is undefined
  - Can register single listener with multiple sources (careful you don't reduce cohesion!)
  - Event listeners implement *event handling methods*
    - are passed an instance of an *event object* (an instance of `java.awt.Event`)
    - contains info. about the event and a reference to the event source

Diagram of Event Interaction

1. A listener is registered with an event source [e.g button.addActionListener(new ActionListener())]
2. The user interacts with the event source (e.g. clicks button with mouse)
3. The event source instantiates an event object containing information about the event
4. The event source passes this event object to the event listener (i.e. invokes the event handler method of its registered listener and passes it the event object as a parameter)



Components as Event Sources

- All Java components inherit the following listener registration methods (addXYZListener) from their parent class java.awt.Component:
  - `void addFocusListener(FocusListener)`
  - `void addKeyListener(KeyListener)`
  - `void addMouseListener(ComponentListener)`
  - `void addMouseMotionListener(MouseMotionListener)`
  - `void addInputMethodListener(InputMethodListener)`
  - `void addComponentListener(ComponentListener)`

- Each AWT component also has specialised listener registration methods:
  - e.g. MenuItem and Button have `addActionListener` methods
  - Usage: `Source.addXYZListener(XYZListener listener)`
  - see table on next slide for complete list

Listeners

- java.awt.event package defines many interfaces for different types of listeners.
- Each interface defines methods that are called when a specific event occurs e.g.
  - `ActionListener` defines a lone method:
    `void actionPerformed(ActionEvent)`
  - invoked when an action event occurs within the source (component) that registered the `ActionListener` with `Component.addActionListener(ActionListener l))`
- Instances of classes that are to act as event listeners must implement one or more of the listener interfaces

Event Objects

- `EventObject` is a simple class
  - base class for all of the `AWTEvent` classes
  - keeps track of its event source, i.e. the object that triggered the event
  - use `EventObject.getSource()` method to return the event source [must be converted from Object to correct type e.g Window `w=(Window)e.getSource()`]
- Declare constants to provide additional info about the event
  - `FOCUS_LOST, FOCUS_GAINED` etc.
  - use methods such as `AWTEvent.getID()` to return constants
- Provide methods to access event properties
  - e.g ActionEvent class
    - `String getActionCommand();`
    - `int getModifiers();`

Event Handling Example

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class ButtonTest extends Applet {
    public void init() {
        Button button = new Button("Press Me");
        button.addMouseListener(new
                    ButtonMouseListener());
        add(button);
    }
}
```

Event Handling Example (cont)

```
class ButtonMouseListener implements MouseListener {
    public void mouseEntered(MouseEvent event) {
        System.out.println("Mouse Entered Button");
    }
    public void mouseExited(MouseEvent event) {
        System.out.println("Mouse Exited Button");
    }
    public void mousePressed (MouseEvent event) { }
    public void mouseClicked (MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
}
```


AWT Adapters

- Most AWT Listener interfaces provide more than one method which must be implemented
- Can be tedious if only one or two are of interest
- e.g. interested only in `WindowListener.windowClosing()` but not the other six Window Events
- Consequently AWT provides Adapters for each of the Listener interfaces that provide multiple methods
- Adapters provide no-op (empty) implementations of the listeners
- => Specific methods can be overridden and handled and the others ignored
- Adapters are classes not interfaces (i.e. they are extended not implemented) therefore only one adapter can be used per class

Adapter Example

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class ButtonTest2 extends Applet {
  public void init() {
      Button button = new Button("Press Me");
      button.addMouseListener(new
             ButtonMouseListener());
      add(button);
  }
}
```


Adapter Example (contd.)

```
class ButtonMouseListener extends MouseAdapter {
  public void mouseEntered(MouseEvent event) {
      System.out.println("Mouse Entered Button");
  }
  public void mouseExited(MouseEvent event) {
      System.out.println("Mouse Exited Button");
  }
}
```

Focus Events

- Maximum of one component can have focus at any one time
- Keyboard events are sent to the focused component
- Typically have a prominent appearance
- Qualified as `FOCUS_LOST` or `FOCUS_GAINED` events
- Focus event can be permanent or temporary
  - permanent focus event is when focus is deliberately changed
  - temporary event occurs when containing object (usually window) loses focus
  - `FocusEvent.isTemporary()`
- Gain focus by:
  - interacting with the component
  - invoking `Component.requestFocus()`
  - typing TAB or SHIFT-TAB
- Use `FocusEvent` and `FocusListener` to handle the focus

Key Events

- Key Events fired when a key is pressed or released in a component that has the focus
- KeyListener interface has KeyPressed(), KeyReleased() and keyTyped() methods
- Each key on the keyboard has a unique key code
  - Returned by `KeyEvent.getKeyCode()` (e.g. `KeyEvent.VK_A`, `KeyEvent.VK_F1` etc. - see API docs for full list of key codes)
  - For key typed events, keyCode is `KeyEvent.VK_UNDEFINED`
- Keys also map to a UNICODE (16 bit) character
  - `KeyEvent.getKeyChar()`
  - Returns the character associated with the key in this event. For example, the key-typed event for shift + "a" returns the value for "A"
  - If no valid Unicode character exists for the key then it returns `KeyEvent.CHAR_UNDEFINED`

Mouse and Mouse Motion Events

- Mouse events and mouse motion events are distinguished via separate `MouseListener` and `MouseMotionListener` interfaces
- Mouse moved and mouse dragged are mouse motion events
- The remaining events (enter/exit, pressed/released and clicked) are mouse events
- Both share the same event class MouseEvent
  - can identify position of cursor: `getPoint()` or `getX(), getY()`
  - can determine key modifiers if any (SHIFT, CTRL etc.) `getModifiers()`
  - can determine click count (can be more than two!) `getClickCount()`

Window Events

- Fired by instances of `java.awt.Window` (and subclasses such as `Frame` and `Dialog`)
- Signify that a window has been *activated/deactivated, iconified/deiconified, opened/closed*, or closing
- `windowClosing()` is commonly overridden for a main frame window to exit the associated application when the frame is closed`